Path Optimization in Programs and its Application to Debugging ⋆

Akash Lal, Junghee Lim, Marina Polishchuk, and Ben Liblit

Computer Sciences Department
University of Wisconsin-Madison
{akash, junghee, mpoli, liblit}@cs.wisc.edu

Abstract. We present and solve a path optimization problem on programs. Given a set of program nodes, called critical nodes, we find a shortest path through the program's control flow graph that touches the maximum number of these nodes. Control flow graphs over-approximate real program behavior; by adding dataflow analysis to the control flow graph, we narrow down on the program's actual behavior and discard paths deemed infeasible by the dataflow analysis. We derive an efficient algorithm for path optimization based on weighted pushdown systems. We present an application for path optimization by integrating it with the Cooperative Bug Isolation Project (*CBI*), a dynamic debugging system. CBI mines instrumentation feedback data to find suspect program behaviors, called bug predictors, that are strongly associated with program failure. Instantiating critical nodes as the nodes containing bug predictors, we solve for a shortest program path that touches these predictors. This path can be used by a programmer to debug his software. We present some early experience on using this hybrid static/dynamic system for debugging.

1 Introduction

Static analysis of programs has been used for a variety of purposes including compiler optimizations, verification of safety properties, and improving program understanding. Static analysis has the advantage of considering all possible executions of a program, thus giving strong guarantees on the program's behavior. In this paper, we present a static analysis technique for finding a program execution sequence that is optimal with respect to some criteria. Given a set of program locations, which we call *critical nodes*, we find a trace among all possible program execution traces that touches the maximum number of these critical nodes and has the shortest length among all such traces. Since reachability in programs is undecidable in general, we over-approximate the set of all possible traces through a program by considering all paths in its control flow graph, and solve the optimization problem on this collection of paths. We also consider how to more closely approximate actual program behavior by discarding paths in the control flow graph deemed infeasible by dataflow analysis [1]. We show that the powerful framework of weighted pushdown systems [2] can be used to represent and solve several variations of the path optimization problem.

⋆ Supported in part by ONR under contracts N00014-01-1-0796 and N00014-01-1-0708, NSF under grant CCR-0305387, and a gift from Microsoft Research.

Why is it important to find paths? Consider the program fragment shown in Figure 1 and suppose that it crashes on some input at line "a[i][j]++". While debugging the program, we find out (using some analysis) that only the statement "a[i] = NULL" in clear() could have caused a null-pointer deference at the crash site. However, looking at this line in isolation gives no indication of what the actual bug is. When we construct a path in the program from the entry point of main() to the crash site that visits this suspect line in clear() we get a path that touches statements shown in bold in Figure 1. It shows that the program can call clear() from process() and then continue execution onto the crash site. Closer examination of this path may suggest that the break statement after clear() should have been a return statement. Seeing paths allows a richer understanding of program behavior than merely examining isolated statements or procedures.

```
int **a;

void main() {
  init(a);
  ...
  process(a);
  ...
}

void clear(int **a) {
  for(...)
    a[i] = NULL;
}

void process(int **a) {
  switch(getchar()) {
  case 'e' :
    clear(a);
    break;
  case 'p' :
    ...
  }
  ...
  a[i][j]++;
}
```

Fig. 1: A buggy program fragment

We have implemented our path optimization algorithm and integrated it with the Cooperative Bug Isolation Project (*CBI*) [3] to create the BTRACE debugging support tool. CBI adds lightweight dynamic instrumentation to software to gather information about runtime behavior. Using this data, it identifies suspect program behaviors, called *bug predictors*, that are strongly associated with program failure. Bug predictors expose the causes and circumstances of failure, and have been used successfully to find previously unknown bugs [4]. CBI is primarily a dynamic system based on mining feedback data from observed runs. Our work on BTRACE represents the first major effort to combine CBI's dynamic approach with static program analysis.

BTRACE enhances CBI output by giving more context for interpreting bug predictors. Using CBI bug predictors as our set of critical nodes, we construct a path from the entry point of the program to the failure site that touches the maximum number of these predictors. CBI associates a numerical score with each bug predictor, with higher scores denoting stronger association with failure. We therefore extend BTRACE to find a shortest path that maximizes the sum of the scores of the predictors it touches. That is, BTRACE finds a path such that the sum of predictor scores of all predictors on the path is maximal, and no shorter path has the same score. We also allow the user to restrict attention to paths that have unfinished calls exactly in the order they appear in a stack trace left behind by the failed program, and to impose constraints on the order in which predictors can be touched. These constraints enhance the utility of BTRACE for debugging purposes by producing a path that is close enough to the actual failing execution of the program to give the user substantial insight into the root causes of failure. We present experimental results in Section 4 to support this claim.

Under the extra constraints described above, the path optimization problem solved by BTRACE can be stated as follows:

THE BTRACE PROBLEM. *Given the control flow graph (N,E) of a program having nodes N and edges E; a single node $n_f \in N$ (representing the crash site of a program); a set of critical nodes $B \subseteq N$ (representing the bug predictors); and a function $\mu : B \longrightarrow \mathbb{R}$ (representing predictor scores), find a path in the control flow graph that first maximizes $\sum_{n \in S} \mu(n)$ where $S \subseteq B$ is the set of critical nodes that the path touches and then minimizes its length. Furthermore, restrict the search for this optimal path to only those paths that satisfy the following constraints:*

1. ***Stack trace.*** *Given a stack trace, consider only those paths that reach n_f with unfinished calls exactly in the order they appear in the stack trace.*
2. ***Ordering.*** *Given a list of node pairs (n_i, m_i) where $n_i, m_i \in B$ and $0 \leq i \leq k$ for some k, consider only those paths that do not touch node m_i before node n_i.*
3. ***Dataflow.*** *Given a dataflow analysis framework, consider only those paths that are not ruled out as infeasible by the dataflow analysis. The requirements on the dataflow analysis framework are specified in Section 3.4.*

Finding a feasible path through a program when one exists is, in general, undecidable. Therefore, even with powerful dataflow analysis, BTRACE can return a path that will never appear in any real execution of the program. We consider this acceptable as we judge the usefulness of a path by how much it helps a programmer debug her program, rather than its feasibility.

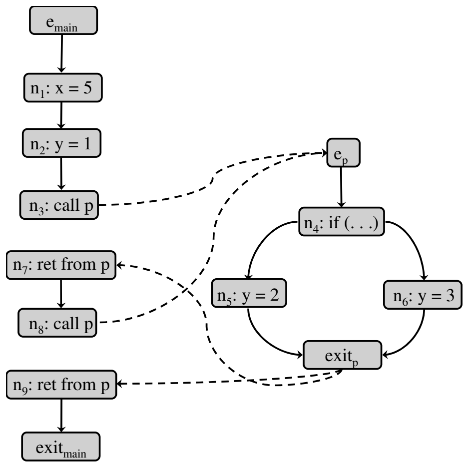
The key contributions of this paper are as follows:

– We present an algorithm that optimizes path selection in a program according to the criteria described above. We use weighted pushdown systems to provide a common setting under which all of the mentioned optimization constraints can be satisfied.
– We describe a hybrid static/dynamic system that combines optimal path selection with CBI bug predictors to support debugging.

The remainder of the paper is organized as follows: Section 2 presents a formal theory for representing paths in a program. Section 3 derives our algorithm for finding an optimal path. Section 4 considers how path optimization can be used in conjunction with CBI for debugging programs and presents experimental results demonstrating that the approach is feasible. Section 5 discusses some of the related work in this area, and Section 6 concludes with some final remarks.

2 Describing Paths in a Program

This section introduces the basic theory behind our approach. In Section 2.1, we formalize the set of paths in a program as a pushdown system. Section 2.2 introduces weighted pushdown systems that have the added ability to associate a value with each path.



(a)

$$
\begin{aligned}
r_1 &= \langle p, e_{main} \rangle \hookrightarrow \langle p, n_1 \rangle \\
r_2 &= \langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle \\
r_3 &= \langle p, n_2 \rangle \hookrightarrow \langle p, n_3 \rangle \\
r_4 &= \langle p, n_3 \rangle \hookrightarrow \langle p, e_p\ n_7 \rangle \\
r_5 &= \langle p, n_7 \rangle \hookrightarrow \langle p, n_8 \rangle \\
r_6 &= \langle p, n_8 \rangle \hookrightarrow \langle p, e_p\ n_9 \rangle \\
r_7 &= \langle p, n_9 \rangle \hookrightarrow \langle p, exit_{main} \rangle \\
r_8 &= \langle p, exit_{main} \rangle \hookrightarrow \langle p, \varepsilon \rangle \\
r_9 &= \langle p, e_p \rangle \hookrightarrow \langle p, n_4 \rangle \\
r_{10} &= \langle p, n_4 \rangle \hookrightarrow \langle p, n_5 \rangle \\
r_{11} &= \langle p, n_4 \rangle \hookrightarrow \langle p, n_6 \rangle \\
r_{12} &= \langle p, n_5 \rangle \hookrightarrow \langle p, exit_p \rangle \\
r_{13} &= \langle p, n_6 \rangle \hookrightarrow \langle p, exit_p \rangle \\
r_{14} &= \langle p, exit_p \rangle \hookrightarrow \langle p, \varepsilon \rangle
\end{aligned}
$$

(b)

Fig. 2: (a) A control flow graph. The *e* and *exit* nodes represent entry and exit points of procedures, respectively. Dashed edges represent interprocedural control flow. (b) A pushdown system that models the control flow graph shown in (a). It uses a single state p and has one rule per CFG edge. Rules r_4 and r_6 correspond to procedure calls and save the return site on the stack. Rules r_8 and r_{14} simply pop-off the top of the stack to reveal the most recent return site.

2.1 Paths in a Program

A control flow graph (CFG) of a program is a graph where nodes are program statements and edges represent possible flow of control between statements. Figure 2a shows the CFG of a program with two procedures. We adopt the convention that each procedure call in the program is represented by two nodes: one is the source of an interprocedural call edge to the callee's entry node and the second is the target of an interprocedural return edge from the callee's exit node back to the caller. In Figure 2a, nodes n_3 and n_7 represent one call from *main* to p; nodes n_8 and n_9 represent a second call.

Not all paths (sequences of nodes connected by edges) in the CFG are valid. For example, the path $[e_{main}\ n_1\ n_2\ n_3\ e_p\ n_4\ n_5\ exit_p\ n_9]$ is invalid because the call at node n_3 should return to node n_7, not node n_9. In general, the valid paths in a CFG are described by a context-free language of matching call/return pairs: for each call, only the matching return edge can be taken at the exit node. For this reason, it is natural to use pushdown systems to describe valid paths in a program [2, 5].

Definition 1. *A **pushdown system** (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ of finite sets where P is the set of states, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A rule $r = (p, \gamma, q, u) \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle q, u \rangle$.*

A PDS is a finite automaton with a stack (Γ^*). It does not take any input, as we are interested in the transition system it describes, not the language it generates.

Definition 2. *A* ***configuration*** *of a pushdown system $\mathscr{P} = (P,\Gamma,\Delta)$ is a pair $\langle p,u\rangle$ where $p \in P$ and $u \in \Gamma^*$. The rules of the pushdown system describe a* ***transition relation*** *$\Rightarrow$ on configurations as follows: if $r = \langle p,\gamma\rangle \hookrightarrow \langle q,u\rangle$ is some rule in Δ, then $\langle p,\gamma u'\rangle \Rightarrow \langle q,uu'\rangle$ for all $u' \in \Gamma^*$.*

The construction of a PDS to represent paths in a CFG is fairly straightforward [2]. An example is shown in Figure 2b. The transition system of the constructed PDS mimics control flow in the program. A sequence of transitions in the transition system ending in a configuration $\langle p,n_1\,n_2\cdots n_k\rangle$, where $n_i \in \Gamma$, is said to have a *stack trace* of $\langle n_1,\cdots,n_k\rangle$: it describes a path in the CFG that is currently at n_1 and has unfinished calls corresponding to the return sites $n_2,\cdots,n_k$. In this sense, a configuration stores an abstract run-time stack of the program, and the transition system describes valid changes that the program can make to it.

2.2 Weighted Pushdown Systems

A weighted pushdown system (WPDS) is obtained by associating a *weight* with each pushdown rule. The weights must come from a set that satisfies bounded idempotent semiring properties [2, 6].

Definition 3. *A* ***bounded idempotent semiring*** *is a quintuple $(D,\oplus,\otimes,\overline{0},\overline{1})$, where D is a set whose elements are called* ***weights****, $\overline{0}$ and $\overline{1}$ are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that*

1. *$(D,\oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent (i.e., for all $a \in D$, $a \oplus a = a$).*
2. *$(D,\otimes)$ is a monoid with $\overline{1}$ as its neutral element.*
3. *$\otimes$ distributes over $\oplus$, i.e., for all $a,b,c \in D$ we have*
$$a\otimes(b\oplus c) = (a\otimes b)\oplus(a\otimes c) \text{ and } (a\oplus b)\otimes c = (a\otimes c)\oplus(b\otimes c)\ .$$
4. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a\otimes\overline{0} = \overline{0} = \overline{0}\otimes a$.*
5. *In the partial order $\sqsubseteq$ defined by: $\forall a,b \in D,\ a \sqsubseteq b$ iff $a\oplus b = a$, there are no infinite descending chains.*

Definition 4. *A* ***weighted pushdown system*** *is a triple $\mathscr{W} = (\mathscr{P},\mathscr{S},f)$ where $\mathscr{P} = (P,\Gamma,\Delta)$ is a pushdown system, $\mathscr{S} = (D,\oplus,\otimes,\overline{0},\overline{1})$ is a bounded idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each pushdown rule.*

The $\otimes$ operation is used to compute the weight of concatenating two paths and the $\oplus$ operation is used to compute the weight of merging parallel paths. If σ is a sequence of rules $[r_1,r_2,\cdots,r_n] \in \Delta^*$, then define the value of σ as $val(\sigma) = f(r_1)\otimes f(r_2)\otimes\cdots\otimes f(r_n)$. In Definition 3, item 3 is required by WPDSs to efficiently explore all paths, and item 5 is required for termination of WPDS algorithms.

For sets of pushdown configurations S and S', let $path(S,S')$ be the set of all rule sequences that transform a configuration in S to a configuration in S'. Let $n\Gamma^* \subseteq \Gamma^*$ denote the set of all stacks that start with n. Existing work on WPDSs allows us to solve the following problems [2]:

Definition 5. *Let $\mathscr{W} = (\mathscr{P},\mathscr{S},f)$ be a weighted pushdown system with $\mathscr{P} = (P,\Gamma,\Delta)$ and let $c \in P\times\Gamma^*$ be a configuration. The* ***generalized pushdown predecessor*** *(GPP_c)* ***problem*** *is to find for each regular set of configurations $S \subseteq (P\times\Gamma^*)$:*

– $\delta(S) \stackrel{\text{def}}{=} \bigoplus\{\, val(\sigma) \mid \sigma \in path(S,c)\}$
– *a* ***witness set*** *of paths* $\omega(S) \subseteq path(S,c)$ *such that* $\bigoplus_{\sigma\in\omega(S)} val(\sigma) = \delta(S)$.

The ***generalized pushdown successor*** *(GPS_c)* ***problem*** *is to find for each regular set of configurations* $S \subseteq P \times \Gamma^*$*:*

– $\delta(S) \stackrel{\text{def}}{=} \bigoplus\{\, val(\sigma) \mid \sigma \in path(c,S)\}$
– *a* ***witness set*** *of paths* $\omega(S) \subseteq path(c,S)$ *such that* $\bigoplus_{\sigma\in\omega(S)} val(\sigma) = \delta(S)$.

For the above definition, we avoid defining a *regular* set of configurations by restricting S to be either a single configuration $\{c'\}$ or $n\Gamma^*$ for some $n \in \Gamma$. The above problems can be considered as backward and forward reachability problems, respectively. Each aims to find the combine of values of all paths between given pairs of configurations ($\delta(S)$). Along with this value, we can also find a witness set of paths $\omega(S)$ that together justify the reported value for $\delta(S)$. This set of paths is always finite because of item 5 in Definition 3. Note that the reachability problems do not require finding the *smallest* witness set, but the WPDS algorithms always find a finite set.

3 Finding an Optimal Path

In this section we solve the specific BTRACE problem defined in Section 1. We begin by developing a solution to the basic path optimization problem without considering dataflow or ordering constraints and then add them back one by one.

3.1 Creating a WPDS

Let (N,E) be a CFG and $\mathscr{P} = (P,\Gamma,\Delta)$ be a pushdown system representing its paths, constructed as described in Section 2.1. Let $B \subseteq N$ be the set of critical nodes. We will use this notation throughout this section. We now construct a WPDS $\mathscr{W} = (\mathscr{P},\mathscr{S},f)$ that can be solved to find the best path.

For each path, we need to keep track of its length and also the set of critical nodes it touches. Let $V = 2^B \times \mathbb{N}$ be a set whose elements each consist of a subset of B (the critical nodes touched) and a natural number (the length of the path). We want to associate each path with an element of V. This is accomplished by defining a weight, which will summarize a set of paths, as a set of elements from V. The combine operation simply takes a union of the weights, but eliminates an element if there is a better one around, i.e., if there are elements (b,v_1) and (b,v_2), the one with shorter path length is chosen. This drives the WPDS to only consider paths with shortest length. The extend operation takes a union of the critical nodes and sums up path lengths for each pair of elements from the two weights. This reflects the fact that when a path with length v_1 that touches the critical nodes in b_1 is extended with a path of length v_2 that touches the critical nodes in b_2, we get a path of length $v_1 + v_2$ that touches the critical nodes in $b_1 \cup b_2$. The semiring constant $\overline{0}$ denotes an infeasible path, and the constant $\overline{1}$ denotes an empty path that touches no critical nodes and crosses zero graph edges. This is formalized in the following definition.

Definition 6. *Let* $\mathscr{S} = (D,\oplus,\otimes,\overline{0},\overline{1})$ *be a bounded idempotent semiring where each component is defined as follows:*

- *The set of weights D is 2^V, the power set of V.*
- *For $w_1, w_2 \in D$, define $w_1 \oplus w_2$ as $reduce(w_1 \cup w_2)$, where*
  $$reduce(A) = \{(b,v) \in A \mid \nexists (b,v') \in A \text{ with } v' < v\}$$
- *For $w_1, w_2 \in D$, define $w_1 \otimes w_2$ as*
  $$reduce(\{(b_1 \cup b_2, v_1 + v_2) \mid (b_1, v_1) \in w_1, (b_2, v_2) \in w_2\})$$
- *The semiring constants $\overline{0}, \overline{1} \in D$ are $\overline{0} = \emptyset$ and $\overline{1} = \{(\emptyset, 0)\}$.*

To complete the description of the WPDS $\mathcal{W}$, we need to associate each pushdown rule with a weight. If $r = \langle p,n \rangle \hookrightarrow \langle p,u \rangle \in \Delta$, then associate it with the weight $f(r) = \{(\{n\} \cap B, 1)\}$. Whenever the rule r is used, the length of the path is increased by one and the set of critical nodes grows to include n if n is a critical node. It is easy to see that for a sequence of rules $\sigma \in \Delta^*$ that describes a path in the CFG, $val(\sigma) = \{(b,v)\}$ where b is the set of critical nodes touched by the path and v is its length.

3.2 Solving the WPDS

An optimal path can be found by solving the generalized pushdown reachability problems on this WPDS. We consider two scenarios here: when we have the crash site but do not have the stack trace of the crash, and when both the crash site and stack trace are available. We start with just the crash site. Let $n_e \in N$ be the entry point of the program, and $n_f \in N$ the crash site.

Theorem 1. *In $\mathcal{W}$, solving $GPS_{\langle p,n_e \rangle}$ gives us $\delta(n_f \Gamma^*) = \{(b,v) \in V \mid$ there is a path from n_e to n_f that touches exactly the critical nodes in b, and the shortest such path has length v $\}$. Moreover, $\omega(n_f \Gamma^*)$ is a set of paths from n_e to n_f such that there is at least one path for each $(b,v) \in \delta(n_f \Gamma^*)$ that touches exactly the critical nodes in b and has length v.*

The above theorem holds because $paths(\langle p,n_e \rangle, \langle p, n_f \Gamma^* \rangle)$ is exactly the set of paths from n_e to n_f, which may or may not have unfinished calls. Taking a combine over the values of such paths selects, for some subsets $b \subseteq B$, a shortest path that touches exactly the critical nodes in b, and discards the longer ones. The witness set must record paths that justify the reported value of $\delta(n_f \Gamma^*)$. Since the value of a path is a singleton-set weight, it must have at least one path for each member of $\delta(n_f \Gamma^*)$.

When we have a stack trace available as some $s \in (n_f \Gamma^*)$, with n_f being the topmost element of s, we can use either *GPS* or *GPP*.

Theorem 2. *In $\mathcal{W}$, solving $GPS_{\langle p,n_e \rangle}$ ($GPP_{\langle p,s \rangle}$) gives us the following values for $W_\delta = \delta(\langle p,s \rangle)$ ($\delta(\langle p,n_e \rangle)$) and $W_\omega = \omega(\langle p,s \rangle)$ ($\omega(\langle p,n_e \rangle)$): $W_\delta = \{(b,v) \in V \mid$ there is a valid path from n_e to n_f with stack trace s that touches all critical nodes in b, and the shortest such path has length v $\}$. $W_\omega =$ a set of paths from n_e to n_f, each with stack trace s such that there is at least one path for each $(b,v) \in W_\delta$ that touches exactly the critical nodes in b and has length v.*

The above theorem allows us to find the required values using either *GPS* or *GPP*. The former uses forward reachability, starting from n_e and going forward in the program, and the latter uses backward reachability, starting from the stack trace s and going backwards. Appendix A presents a detailed discussion on the complexity of solving these problems on our WPDS. The worst-case complexity is exponential in the number of critical nodes and (practically) linear in the size of the program. The exponential complexity in critical nodes is, unfortunately, unavoidable. The reason is that the path optimization problem we are trying to solve is a strict generalization of the traveling salesman problem: our objective is to find a shortest path between two points that touches a given set of nodes. However, we did not find this complexity to be a limitation in our experiments.

Having obtained the above W_δ and W_ω values, we can find an optimal path easily. Let $\mu : B \rightarrow \mathbb{R}$ be a user-defined measure that associates a score with each critical node. We compute a score for each $(b, v) \in W_\delta$ by summing up the scores of all critical nodes in b and then choose the pair with highest score. Extracting a path corresponding to that pair in W_ω gives us an optimal path. Some advantages of having such a user-defined measure are that the user can specify bug predictor scores given by CBI, or make up his own scores. The user can also give a negative score to critical nodes that should be *avoided* by the path. Critical nodes with zero score can be added and used for specifying ordering constraints (Section 3.3). This lets our tool work interactively with the user to find a suitable path. More generally, we can allow the user to give a measure $\hat{\mu} : (2^B \times \mathbb{N}) \rightarrow \mathbb{R}$ that directly associates a score with a path. Using such a measure, the user can decide to choose shorter paths instead of paths that touch more critical nodes.

3.3 Adding Ordering Constraints

We now add ordering constraints to the path optimization problem. Suppose that we have a constraint "node n must be visited before node m," which says that we can only consider paths that do not visit m before visiting n. It is relatively easy to add such constraints to the WPDS given above. The extend operation is used to compute the value of a path. We simply change it to yield $\overline{0}$ for paths that do not satisfy the above ordering constraint. For $w_1, w_2 \in D$, redefine $w_1 \otimes w_2$ as $reduce(A)$ where

$$A = \{(b_1 \cup b_2, v_1 + v_2) \mid (b_1, v_1) \in w_1, (b_2, v_2) \in w_2, \neg(m \in b_1, n \in b_2)\}$$

If we have more than one ordering constraint, then we simply add more clauses, one for each constraint, to the above definition of extend.

These constraints do not change the worst case asymptotic complexity of solving reachability problems in WPDS. However they do help prune down the paths that need to be explored, because each constraint cuts down on the size of weights produced by the extend operation.

3.4 Adding Dataflow Analysis

So far we have not considered interpreting the semantics of the program other than its control flow. This implies that the WPDS can find infeasible paths: ones that cannot occur in any execution of the program. An example is a path that assigns `x := 1` and then follows the true branch of the conditional `if (x == 0)`. In general, it is undecidable to restrict attention to paths that actually occur in some program execution, but if we can rule out many infeasible paths, we increase the chances of presenting a feasible or near-feasible path to the user. This can be done using dataflow analysis.

Dataflow analysis is carried out to approximate, for each program variable, the set of values that the variable can take at each point in the program. When a dataflow analysis satisfies certain conditions, it can be integrated into a WPDS by designing an appropriate weight domain [2, 5]. Examples of such dataflow analyses include linear constant propagation [7] and affine relation analysis [8, 9]. In particular, we can use any bounded idempotent semiring weight domain $\mathscr{S}_d = (D_d, \oplus_d, \otimes_d, \overline{0}_d, \overline{1}_d)$ provided that when given a function $f_d : \Delta \to D_d$ that associates each PDS rule (CFG edge) with a weight, it satisfies the following property: given any (possibly infinite) set $\Sigma \subseteq \Delta^*$ of paths between the same pair of program nodes, we have $\bigoplus_{\sigma\in\Sigma} val_d(\sigma) = \overline{0}_d$ only if all paths in Σ are infeasible, where $val_d(\ [r_1, \cdots, r_k]\ ) = f_d(r_1) \otimes_d \cdots \otimes_d f_d(r_k)$. In particular this means that $val_d(\sigma) = \overline{0}_d$ only if σ is an infeasible path. This imposes a soundness guarantee on the dataflow analysis: it can only rule out infeasible paths. Details on how classical dataflow analysis frameworks [1] can be encoded as weight domains can be found in Reps et al. [2]. The basic idea is to encode dataflow transformers that capture the effect of executing a program statement, or a sequence of statements, as weights. The extend operation composes transformers and the combine operation takes their *meet* in the dataflow value lattice.

Such a translation from dataflow transformers to a weight domain allows us to talk about the meet-over-all-paths between configurations of a pushdown system. For example, solving $GPS_{\langle p,n_e\rangle}$ on this weight domain gives us $\delta(\langle p, n_1 n_2 \cdots n_k\rangle)$ as the combine (or meet) over the values of all paths from n_e to n_1 that have the stack trace $n_1 n_2 \cdots n_k$. This is a unique advantage that we gain over conventional dataflow analysis by using WPDSs.

Given $\mathscr{S}_d$ and f_d as above, we change the weight domain of our WPDS as follows.

Definition 7. *Let $\mathscr{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ be a bounded idempotent semiring where each component is defined as follows:*

- *The set of weights D is $2^{2^B \times \mathbb{N} \times D_d}$, the power set of the set $2^B \times \mathbb{N} \times D_d$.*
- *For $w_1, w_2 \in D$, define $w_1 \oplus w_2$ as $reduce_d(w_1 \cup w_2)$ where $reduce_d(A)$ is defined as $\{(b, min\{v_1, \cdots v_n\}, d_1 \oplus_d \cdots \oplus_d d_n) \mid (b, v_i, d_i) \in A, 1 \le i \le n\}$*
- *For $w_1, w_2 \in D$, define $w_1 \otimes w_2$ as $reduce_d(A)$ where A is the set*
$$\left\{ (b_1 \cup b_2, v_1 + v_2, d_1 \otimes_d d_2) \;\middle|\; \begin{array}{l} (b_1, v_1, d_1) \in w_1, (b_2, v_2, d_2) \in w_2, d_1 \otimes_d d_2 \\ \neq \overline{0}_d, (b_1, b_2) \textit{ satisfy all ordering constraints} \end{array} \right\}$$
- *The semiring constants $\overline{0}, \overline{1} \in D$ are $\overline{0} = \emptyset$ and $\overline{1} = \{(\emptyset, 0, \overline{1}_d)\}$.*

Here (b_1, b_2) satisfy all ordering constraints iff for each constraint “visit n before m,” it is not the case that $m \in b_1$ and $n \in b_2$.

The weight associated with each rule $r = \langle p, n\rangle \hookrightarrow \langle p, u\rangle \in \Delta$ is given by $f(r) = \{(\{n\} \cap B, 1, f_d(r))\}$. Each path is now associated with the set of predictors it touches, its length, and its dataflow value. Infeasible paths are removed during the extend operation as weights with dataflow value $\overline{0}_d$ are discarded. More formally, for a path $\sigma \in \Delta^*$ in the CFG, $val(\sigma) = \{(b, v, w_d)\}$ if $w_d = val_d(\sigma) \neq \overline{0}_d$ is the dataflow value associated with the path, v is the length of the path, b is the set of critical nodes touched by the path, and the path satisfies all ordering constraints. If σ does not satisfy some ordering constraint or if $val_d(\sigma) = \overline{0}_d$, then $val(\sigma) = \emptyset = \overline{0}$. Analysis using this weight domain is similar to the “property simulation” used in ESP [10], where a distinct dataflow value is maintained for each property-state. We maintain a distinct dataflow weight for each subset of critical nodes.

Instead of repeating Theorems 1 and 2, we just present the case of using GPS when the stack trace $s \in (n_f \Gamma^*)$ is available. Results for other cases can be obtained similarly.

Theorem 3. *In the WPDS obtained from the weight domain defined in Definition 7, solving $GPS_{\langle p,n_e\rangle}$ gives us the following values:*

- $\delta(\langle p,s\rangle) = \{(b,v,w_d) \mid$ *there is a path from n_e to n_f with stack trace s that visits exactly the critical nodes in b, satisfies all ordering constraints, is not infeasible under the weight domain $\mathcal{S}_d$, and the shortest such path has length v* $\}$.
- $\omega(\langle p,s\rangle)$ *contains at least one path for each $(b,v,w_d) \in \delta(\langle p,s\rangle)$ that goes from n_e to n_f with stack trace s, visits exactly the predictors in b, satisfies all ordering constraints and has length v. More generally, for each $(b,v,w_d) \in \delta(\langle p,s\rangle)$ it will have paths $\sigma_i, 1 \leq i \leq k$ for some constant k such that $val(\sigma_i) = \{(b,v_i,w_i)\}$, $min\{v_1,\cdots,v_k\} = v$, and $w_1 \oplus_d \cdots \oplus_d w_k = w_d$.*

The worst case time complexity in the presence of dataflow analysis increases by a factor of $H_d(C_d + E_d)$ where H_d is height of $\mathcal{S}_d$, C_d is time required for applying $\oplus_d$, and E_d is the time required for applying $\otimes_d$.

Theorem 3 completely solves the BTRACE problem mentioned in Section 1. The next section presents the dataflow weight domain that we used for our experiments.

3.5 Example and Extensions for Using Dataflow Analysis

Copy Constant Propagation We now give an example of a weight domain that can be used for dataflow analysis. We encode copy-constant propagation [11] as a weight domain. A similar encoding is used by Sagiv, Reps, and Horwitz [7]. Copy-constant propagation aims to determine if a variable has a fixed constant value at some point in the program. It interprets constant-to-variable assignments (`x := 1`) and variable-to-variable assignments (`x := y`) and abstracts all other assignments as `x :=` $\perp$, which says that `x` may not have a constant value. We ignore conditions on branches for now.

Let *Var* be the set of all global (integer) variables of a given program. Let $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$ and $(\mathbb{Z}_\perp^\top, \sqcap)$ be the standard constant propagation meet semilattice obtained from the partial order $\perp \sqsubseteq_{cp} c \sqsubseteq_{cp} \top$ for all $c \in \mathbb{Z}$. Then the set of weights of our weight domain is $D_d = Var \rightarrow (2^{Var} \times \mathbb{Z}_\perp^\top)$. Here, $\tau \in D_d$ represents a dataflow transformer that summarizes the effect of a executing a sequence of program statements as follows: if $env : Var \rightarrow \mathbb{Z}$ is the state of the program before the statements are executed and $\tau(\mathtt{x}) = (\{\mathtt{x}_1,\cdots,\mathtt{x}_n\}, c)$ for $c \in \mathbb{Z}_\perp^\top$, then the value of a variable `x` after the statements are executed is $env(\mathtt{x}_1) \sqcap env(\mathtt{x}_2) \cdots \sqcap env(\mathtt{x}_n) \sqcap c$. Let $\tau^{\mathtt{v}}(\mathtt{x})$ be the first component of $\tau(\mathtt{x})$ and $\tau^{\mathtt{c}}(\mathtt{x})$ be the second component. Then we can define the semiring operations as follows: the combine operation is a concatenation of expressions and the extend operation is substitution. Formally, for $\tau_1, \tau_2 \in D_d$,

$$\tau_1 \oplus_d \tau_2 = \lambda\mathtt{x}.(\tau_1^{\mathtt{v}}(\mathtt{x}) \cup \tau_2^{\mathtt{v}}(\mathtt{x}), \tau_1^{\mathtt{c}}(\mathtt{x}) \sqcap \tau_2^{\mathtt{c}}(\mathtt{x}))$$

$$\tau_1 \otimes_d \tau_2 = \lambda\mathtt{x}.(\bigcup_{\mathtt{y}\in\tau_2^{\mathtt{v}}(\mathtt{x})} \tau_1^{\mathtt{v}}(\mathtt{y}), \tau_2^{\mathtt{c}}(\mathtt{x}) \sqcap (\bigsqcap_{\mathtt{y}\in\tau_2^{\mathtt{v}}(\mathtt{x})} \tau_1^{\mathtt{c}}(\mathtt{y})))$$

The semiring constants are given by $\overline{0}_d = \lambda\mathtt{x}.(\emptyset, \top)$ and $\overline{1}_d = \lambda\mathtt{x}.(\{\mathtt{x}\}, \top)$.

Handling Conditionals Handling branch conditions is problematic because dataflow analysis in the presence of conditions is usually very hard. For example, finding whether a branch condition can ever evaluate to true, even for copy-constant propagation, is

PSPACE-complete [12]. Therefore, we have to resort to approximate dataflow analysis, i.e., we give up on computing meet-over-all-paths. This translates into relaxing the distributivity requirement on the weight domain $\mathcal{S}_d$. Fortunately, WPDSs can handle non-distributive weight domains [2] by relaxing Definition 3 item 3 as follows. If D is the set of weights, then for all $d_1, d_2, d_3 \in D$,

$$d_1 \otimes (d_2 \oplus d_3) \sqsubseteq (d_1 \otimes d_2) \oplus (d_1 \otimes d_3); \quad (d_1 \oplus d_2) \otimes d_3 \sqsubseteq (d_1 \otimes d_3) \oplus (d_2 \otimes d_3)$$

where $\sqsubseteq$ is the partial order defined by $\oplus$: $d_1 \sqsubseteq d_2$ iff $d_1 \oplus d_2 = d_1$. Under this weaker property, the generalized reachability problems can only be solved approximately, i.e., instead of obtaining $\delta(c)$ for a configuration c, we only obtain a weight w such that $w \sqsubseteq \delta(c)$. For our path optimization problem, this inaccuracy will be limited to the dataflow analysis. We would only eliminate some of the paths that the dataflow analysis can find infeasible and might find a path σ such that $val_d(\sigma) = \overline{0}_d$. This is acceptable because it is not possible to rule out all infeasible paths anyway. Moreover, it allows us the flexibility of putting in a simple treatment for conditions in most dataflow analyses. The disadvantage is that we lose a strong characterization of the type of paths that will be eliminated.

For copy-constant propagation, we extend the set of weights by $\{\rho_e \mid e \text{ is an arithmetic condition}\}$. We associate weight ρ_e with the rule $\langle p, n\rangle \hookrightarrow \langle p, m\rangle$ ($n, m \in \Gamma$) if the corresponding CFG edge can only be taken when e evaluates to true on the program state at n. For example, we associate the weight $\rho_{\mathtt{x}=0}$ with the true branch of the conditional `if (x == 0)` and weight $\rho_{\mathtt{x}\neq 0}$ with its false branch. The extend operation is modified such that for $\tau \in D_d$, $\tau \otimes \rho_e$ evaluates the condition e under the information provided by τ and results to $\overline{0}_d$ if e evaluates to false. Otherwise, the extend is simply τ. More details can be found in a companion technical report [13].

Handling Local Variables A recent extension to WPDSs [5] shows how local variables can be handled by using *merge functions* that allow for local variables to be saved before a call and then merged with the information returned by the callee to compute the effect of the call. This treatment for local variables allows us to restrict each weight to manage the local variables of only one procedure. Details of the construction of these merge functions are given in a companion technical report [13].

4 Integrating BTRACE and CBI

The formalisms of Section 3 may be used for solving a variety of optimization problems concerned with touching key program points along some path. BTRACE represents one application of these ideas: an enhancement to the statistical debugging analysis performed by the Cooperative Bug Isolation Project (CBI).

4.1 A Need for Failure Paths

CBI uses runtime instrumentation and statistical modeling techniques to diagnose bugs in widely deployed software. CBI identifies suspect program behaviors, called *bug predictors*, that are strongly associated with program failure. Candidate behaviors may include branch directions, function call results, values of variables, and other dynamic properties [14]. Each bug predictor is assigned a numerical score in $\mathbb{R}^+$ that balances two key factors: (1) how much this predictor increases the probability of failure, and (2) how many failed runs this predictor accounts for. Thus, high-value predictors warrant close examination both because they are highly correlated with failure and because they account for a large portion of the overall failure rate seen by end users [4].

A key strength of CBI is that it samples behavior for the entire dynamic lifetime of a run; however, interpreting the resulting predictors, which may be located anywhere in the program prior to the failure point, can be very challenging. Rather than work with isolated bug predictors, the programmer would like to navigate forward and backward along the path that led to failure. BTRACE constructs a path that hits several high-ranked predictors. This can help the programmer draw connections between sections of code that, though seemingly unrelated, act in concert to bring the program down.

4.2 BTRACE Implementation

We have implemented BTRACE using the WPDS++ library [15]. To manage the exponential complexity in the number of bug predictors, we efficiently encode weights using abstract decision diagrams (ADDs) provided by the CUDD library [16]. Additional details on how the semiring operations are implemented on ADDs may be found in a companion technical report [13].

A BTRACE debugging session starts with a list of related bug predictors, believed by CBI to represent a single bug. We designate this list (or some high-ranked prefix thereof) as the critical nodes and insert them at their corresponding locations in the CFG. Branch predictors, however, may be treated as a special case. These predictors associate the direction of a conditional with failure, and therefore can be repositioned on the appropriate branch. This can be seen as one example of exploiting not just the location but also the semantic meaning of a bug predictor; branch predicates make this easy because their semantic meaning directly corresponds to control flow.

For dataflow analysis, we track all integer- and pointer-valued variables and structure fields. We do not track the contents of memory and any write to memory via a pointer is replaced with assignments of $\perp$ to all variables whose address was ever taken. Direct structure assignments are expanded into component-wise assignments to all fields of the structure.

4.3 Case Studies: Siemens Suite

We have applied BTRACE to three buggy programs from the Siemens test suite [17]: TCAS v37, REPLACE v8, and PRINT_TOKENS2 v6. These programs do not crash; they merely produce incorrect output. Thus our analysis is performed without a stack trace, instead treating the exit from `main()` as the "failure" point. We find that BTRACE can be useful even for non-fatal bugs.

TCAS has an array index error in a one-line function that contains no CBI instrumentation and thus might easily be overlooked. Without bug predictors, BTRACE produces the shortest possible path that exits `main()`, revealing nothing about the bug. After adding the top-ranked predictor, BTRACE isolates lines with calls to the buggy function.

REPLACE has an incorrect function return value. BTRACE with the top two predictors yields a path through the faulty statement. Each predictor is located within one of two disjoint chains of function calls invoked from `main()`, and neither falls in the same function as the bug. Thus, while the isolated predictors do not directly reveal the bug, the BTRACE failure path through these predictors does.

PRINT_TOKENS2 has an off-by-one error. Again, two predictors suffice to steer BTRACE to the faulty line. Repositioning of branch predictors is critical here. Even with all nineteen CBI-suggested predictors and dataflow analysis enabled, a correct failure path only results if branch predictors are repositioned to steer the path in the proper direction.

4.4 Case Studies: CCRYPT and BC

We have also run BTRACE on two small open source utilities: CCRYPT v1.2 and BC v1.06. CCRYPT is an encryption/decryption tool and BC is an arbitrary precision calculator. Both are written in C. Fatal bugs in each were first characterized in prior work by Liblit et al. [14]. More detailed discussion of experimental results can be found in a companion technical report [13].

CCRYPT has an input validation bug. Reading end-of-file yields a `NULL` string (`char *`) that is subsequently dereferenced without being checked first. If given only a stack trace, BTRACE builds an infeasible path that takes several impossible shortcuts through initialization code. These shortcuts also yield `NULL` values, but in places that are properly checked before use and therefore cannot be the real bug. The path remains the same if we add dataflow analysis (but no bug predictors), or if we add up to fourteen bug predictors (but no dataflow analysis).

However, if BTRACE uses both dataflow analysis and at least eleven bug predictors, the failure path changes to a feasible path that correctly describes the bug: non-`NULL` values in the well-checked initialization code, and a fatal unchecked `NULL` value later on. This feasible path also arises from just a stack trace if one manually inserts ordering constraints to require that bug predictors appear after initialization code, e.g. if the initialization code were assumed to be bug-free. The combination of dataflow analysis and bug predictors make such manual, a priori assumptions unnecessary.

BC has a buffer overrun: a bad loop index in `more_arrays()` silently trashes memory. The program keeps running but may eventually crash during a subsequent call to `bc_malloc()`. The stack trace at the point of failure suggests heap corruption but provides no real clues as to when the corruption occurred or by what piece of code. CBI-identified bug predictors are scattered across several files and their relationship may not be clear on first examination.

Using one bug predictor, BTRACE builds a path that calls `more_arrays()` early in execution. This path is feasible but misleading: `more_arrays()` is always called early in execution, and only a second or subsequent call to `more_arrays()` can cause failure. Using two or more bug predictors forces the path to include a fatal second call to `more_arrays()`, correctly reflecting the true bug. By reading in-progress calls out of the failure trace, we can easily reconstruct the entire stack at the call to `more_arrays()` or any other point of interest and thereby give deeper context to the frontier of the bad code.

CBI actually produces two ranked lists of related bug predictors for BC, suggesting two distinct bugs. BTRACE produces the same path using either list, suggesting that they correspond to a single bug. BTRACE is correct: the two lists do correspond to a single bug. CBI can be confused by sampling noise, statistical approximation, incompleteness of dynamic data, and other factors. BTRACE is a useful second check, letting us unify equivalent bug lists that CBI has incorrectly held apart.

Section 3.2 mentioned that solving the WPDS may require time exponential in the number of bug predictors. We find that the actual slowdown is gradual and that the absolute performance of BTRACE is good. As expected, the *GPS* phase dominates; creating the initial WPDS and extracting a witness path from the solved system take negligible time. The small CCRYPT application has 13,661 CFG nodes, with about 1,300 on a typical failure path. BTRACE requires 0.10 seconds to find a path using zero CCRYPT predictors, increasing gradually to 0.97 seconds with fifteen predictors. Adding more predictors slows the analysis gradually, amplified only when adding a predictor forces BTRACE to build a longer failure path. BC is larger at 45,234 CFG nodes, and a typical failure path produced by BTRACE is about 3,000 nodes long. The complete analysis takes from two to four seconds with up to four predictors.

Adding dataflow analysis slows the analysis by a factor of between four and twelve, depending on configuration details. Analysis with dataflow and realistic numbers of bug predictors takes about thirteen seconds for BC and less than two seconds for CCRYPT.

5 Related Work

The CodeSurfer Path Inspector tool [18, 19] uses weighted pushdown systems for verification: to see if a program can drive an automaton, summarizing a program property, into a bad state. If this is possible, it uses witnesses to produce a faulty program path. It can also use dataflow analyses by encoding them as weights to rule out infeasible paths. We use WPDSs for optimizing a property instead of verifying it, which has not been previously explored.

Liblit and Aiken directly consider the problem of finding likely failure paths through a program [20]. They present a family of analysis techniques that exploit dynamic information such as failure sites, stack traces, and event logs to construct the set of possible paths that a program might have taken. They could not, however, optimize path length or the number of events touched when all of them might be unreachable in a single path. Our approach is, therefore, more general. BTRACE incorporates these techniques, along with dataflow analysis, within the unifying framework of weighted pushdown systems. Another difference is that instead of using event logs, we use the output of CBI to guide the path-finding analysis. The theory presented in Section 3 can be extended to incorporate event logs by adding ordering constraints to appropriately restrict the order in which events must be visited by a path.

PSE is another tool for finding failing paths [21]. It requires a user-provided description of how the error could have occurred, e.g., "a pointer was assigned the value `NULL`, and then dereferenced." This description is in the form of a finite state automaton, and the problem of finding a failing run is reduced to finding a backward path that drives this automaton from its *error* state to its initial state. PSE solves this in the presence of pointer-based data structures and aliasing. Our work does not require any user description of the bug that might have caused the crash, but we do not yet handle pointer-based structures. Like PSE, we can use pointer analysis as a preprocessing step to produce more accurate dataflow weights.

In Definitions 6 and 7, we define semirings that are the power set of the values we want to associate with each path. This approach has been presented in a more general setting by Lengauer and Theune [22]. The power set operation is used to add distributivity to the semiring, and a reduction function, such as our *reduce*, ensures that we never form sets of more elements than necessary.

Our lists of bug predictors are derived using the iterative ranking and elimination algorithm of Liblit et al. [4]. Several other statistical debugging algorithms for CBI-style data have been proposed, including ones based on regularized curve fitting [23], sparse disjunction learning [24], probability density function estimation [25], support vector machines [26], and random forests [26]. BTRACE path reconstruction can use predictors arising from any of these techniques; we require only a list of predictors and numerical scores reflecting their importance. Further study may reveal whether certain statistical debugging algorithms yield more useful BTRACE paths than others.

6 Conclusions

We have presented a static analysis technique to build BTRACE, a tool that can find an optimal path in a program under various constraints imposed by a user. Using bug predictors produced by CBI, BTRACE can perform a postmortem analysis of a program and reconstruct a program path that reveals the circumstances and causes of failure. The paths produced by BTRACE might not be feasible, but we intend for them to help programmers understand the bug predictors produced by CBI and locate bugs more quickly. BTRACE provides user options to supply additional constraints in the form of stack traces and ordering constraints, the latter of which allow the user to guide the tool interactively while locating a bug. Our case studies show that the BTRACE path can isolate the chain of events leading to failure, and, given enough predictors, has the ability to lead the programmer directly to the faulty code. More experiments are required to prove the utility of BTRACE in debugging larger software systems, but initial results look promising.

7 Acknowledgments

We would like to thank Susan Horwitz for her insightful comments on an earlier draft of this paper.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1985)
2. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: Sci. of Comp. Prog. Volume 58. (2005) 206–263
3. Liblit, B.: Cooperative Bug Isolation. PhD thesis, University of California, Berkeley (2004)
4. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: SIGPLAN Conf. on Prog. Lang. Design and Impl. (2005)
5. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: Computer Aided Verification. (2005) 434–448
6. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: Symp. on Princ. of Prog. Lang. (2003) 62–73
7. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comp. Sci. **167** (1996) 131–170
8. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Symp. on Princ. of Prog. Lang. (2004)

9. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: European Symp. on Programming. (2005)
10. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: SIGPLAN Conf. on Prog. Lang. Design and Impl. (2002) 57–68
11. Wegman, M., Zadeck, F.: Constant propagation with conditional branches. In: Symp. on Princ. of Prog. Lang. (1985) 291–299
12. Müller-Olm, M., Rüthing, O.: On the complexity of constant propagation. In: European Symp. on Programming. (2001) 190–205
13. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: BTRACE: Path optimization for debugging. Technical Report 1535, University of Wisconsin-Madison (2005)
14. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: SIGPLAN Conf. on Prog. Lang. Design and Impl. (2003)
15. Kidd, N., Reps, T., Melski, D., Lal, A.: WPDS++: A C++ library for weighted pushdown systems (2005) <http://www.cs.wisc.edu/wpis/wpds++>.
16. Somenzi, F.: Colorado University Decision Diagram package. Technical report, University of Colorado, Boulder (1998)
17. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proc. of the 16th Int. Conf. on Softw. Eng., IEEE Computer Society Press (1994) 191–200
18. GrammaTech, Inc.: CodeSurfer Path Inspector (2005) <http://www.grammatech.com/products/codesurfer/overview_pi.html>.
19. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.H., Teitelbaum, T.: Model checking x86 executables with CodeSurfer/x86 and WPDS++. In: Computer Aided Verfication. (2005)
20. Liblit, B., Aiken, A.: Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, University of California, Berkeley (2002)
21. Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: PSE: explaining program failures via postmortem static analysis. In: Found. Softw. Eng. (2004) 63–72
22. Lengauer, T., Theune, D.: Unstructured path problems and the making of semirings (preliminary version). In: WADS. Volume 519 of Lecture Notes in Computer Science., Springer (1991) 189–200
23. Zheng, A.X., Jordan, M.I., Liblit, B., Aiken, A.: Statistical debugging of sampled programs. In Thrun, S., Saul, L., Schölkopf, B., eds.: Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004)
24. Zheng, A.X.: Statistical Software Debugging. PhD thesis, Univ. of California, Berkeley (2005)
25. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: Statistical mmodel-based bug localization. In: Found. Softw. Eng., New York, NY, USA, ACM Press (2005) 286–295
26. Jiang, L., Su, Z.: Automatic isolation of cause-effect chains with machine learning. Technical Report CSE-2005-32, University of California, Davis (2005)

A Complexity of Solving the WPDS

In this section, we discuss the worst-case running time complexity of solving the WPDS constructed with the weight domain defined in Definition 6. Each of the methods outlined in Theorems 1 and 2 require solving either *GPS* or *GPP* and then reading the value of $\delta(c)$ for some configuration c. We do not consider the time required for reading the witness value as it can be factored into these two steps. Let $|\Delta|$ be the number of pushdown rules (or the size of the CFG), $|\mathrm{Proc}|$ the number of procedures in the program, n_e the entry point of the program, $|B|$ the number of critical nodes, and L the length of a shortest path to the most distant CFG node from n_e. The height (length of the longest descending chain) of the semiring we use is $H = 2^{|B|}L$ and the time required to perform each semiring operation is $T = 2^{|B|}$.

To avoid requiring more WPDS terminology, we specialize the complexity results of solving reachability problems on WPDS [2] to our particular use. $GPS_{\langle p,n_e\rangle}$ can be solved in $O(|\Delta|\ |\mathrm{Proc}|\ H\ T)$ time and $GPP_{\langle p,s\rangle}$ requires $O(|s|\ |\Delta|\ H\ T)$ time. Reading the value of $\delta(\langle p,n_e\rangle)$ is constant time and $\delta(\langle p,s\rangle)$ requires $O(|s|\ T)$ time. We can now put these results together.

When no stack trace is available, the only option is to use Theorem 1. Obtaining an optimal path in this case requires time $O(|\Delta|\ |\mathrm{Proc}|\ 2^{2|B|}\ L)$. When a stack trace is available, Theorem 2 gives us two options. Suppose we have k stack traces available to us (corresponding to multiple failures caused by the same bug). In the first option, we solve $GPS_{\langle p,n_e\rangle}$, and then ask for the value of $\delta(\langle p,s\rangle)$ for each stack trace available. This has worst-case time complexity $O(|\Delta|\ |\mathrm{Proc}|\ 2^{2|B|}\ L\ +\ k\ |s|\ 2^{|B|})$ where $|s|$ is the average length of the stack traces. The second option requires a stack trace s, solves $GPP_{\langle p,s\rangle}$ and then asks for the value of $\delta(\langle p,n_e\rangle)$. This has worst-case time complexity $O(k\ |s|\ |\Delta|\ 2^{2|B|}\ L)$. As is evident from these complexities, the second option should be faster, but its complexity grows faster with an increase in k. Note that these are only worst-case complexities, and comparisons based on them need not hold for the average case. In fact, in WPDS++ [15], the WPDS implementation that we use, solving *GPS* is usually faster than solving *GPP*.[1]

Let us present some intuition into the complexity results stated above. Consider the CFG shown in Figure 3. If node n_2 is a critical node, then a path from n_1 to n_6 that takes the left branch at n_2 has length 4. The path that takes the right branch has length 3, and touches the same critical nodes as the first path. Therefore, at n_6, the first path can be discarded and we only need to remember the second path. In this way, branching in the program, which increases the total number of paths through the program, only increases the complexity linearly ($|\Delta|$). Now, if node n_3 is also a critical node, then at n_6 we need to remember both paths: one touches more critical nodes and the other has shorter length. (For a path that comes in at n_1, and has already touched n_3, it is better to take the shorter right branch at n_2.) In general, we need to remember a path for each subset of the set of all critical nodes. This is reflected in the design of our weight domain and is what contributes to the exponential complexity with respect to the number of critical nodes.

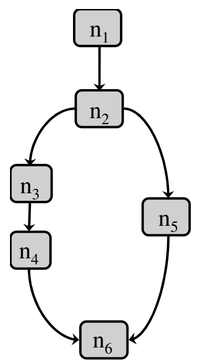


Fig. 3: A simple control flow graph

[1] The implementation does not take advantage of the fact that the PDS has been obtained from a CFG. Backward reachability is easier on CFGs as there is at most one known predecessor of a return-site node.